

Information Systems Institute

Distributed Systems Group (DSG)
VU Distributed Systems Technologies SS 2012 (184.260)

Assignment 1

Submission Deadline: 13.4.2012, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the TUWEL forum) are allowed, but the code has to be written alone. If we find that two students submitted the same, or very similar assignments, these students will be graded with 0 points (no questions asked). We will use automated plagiarism checks to compare solutions. Note that we will also include the submissions of previous years in our plagiarism checks.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution includes all the libraries and dependencies and it compiles without errors. If you are unsure, test to compile your submission on a different computer (e.g., one of the ZID lab computers). **We are known to deduct points drastically if your code does not build/run out of the box.**
- A good introduction into JPA can be found in Part VI of the Java EE 6 Tutorial¹.
- Before asking questions about Hibernate, check out the Hibernate Documentation² page. If you cannot find an answer to your question, consult the Hibernate forum³.
- For the MongoDB tasks, consult the MongoDB Java tutorial⁴.
- Use Java 1.6.14+⁵ and MySQL 5.1+⁶ as your database management system. Additionally, for the last part of the assignment we will need MongoDB 2.0.2⁷.
- For your solution, use the provided project stubs. All needed libraries are already included. If you want to use other libraries, feel free to integrate them. We expect (as can be seen in the persistence-unit configuration of persistence.xml) that you setup a database dst that can be accessed by root (without a password). You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Build scripts for ant⁸ are already included, you may alter them, but the submitted solution has to be compilable and runnable with the predefined targets. Again: make sure that all settings are as expected before you submit!
- For each code task, provide some rudimentary tests to show that your solution works. Complete the dst0x methods in dst1.Main. Remember, the main point of these tests is to illustrate what your solution does, so try to be as clear with your debug output as possible. No debug output is very bad, and too much (e.g., many screen pages) is just as bad. Aim for a good middle ground, which allows a tutor to check your solution quickly.
- **Later assignments will be based on (parts of) the solution of this lab. Keep this in mind while implementing your code.**

¹<http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html>

²<http://hibernate.org/docs>

³<https://forum.hibernate.org/>

⁴<http://www.mongodb.org/display/DOCS/Java+Tutorial>

⁵<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

⁶<http://dev.mysql.com/downloads/mysql/5.1.htm#downloads>

⁷<http://www.mongodb.org/downloads>

⁸<http://ant.apache.org/>

A. Code Part

1. Mapping Persistent Classes and Associations (16 Points)

In this year's lab, it is your task to build the enterprise application for managing high-performance computing grids. A user can create and assign jobs to a grid. The grid then executes the job on a cluster of computers, i.e., on many machines simultaneously. Our simple assumption here is that each job needs a fixed number of CPUs to execute. Jobs may be assigned freely to computers and their CPUs. For instance, we assume that a job which needs to run on 8 CPUs can either run on 2 quad-cores or on 8 single core computer. Computers are organized in clusters. Every computer belongs to exactly one cluster. It is also possible that a cluster is recursively composed of other clusters. Each Cluster is maintained by exactly one administrator.

Please note that your task in this and the following assignments is to build the Grid Management System, not the Grid itself. We will only simulate the actual execution of jobs.

The domain model that we use as starting point is defined in the following UML class diagram. We will come back and slightly extend this data model in future tasks and assignments.

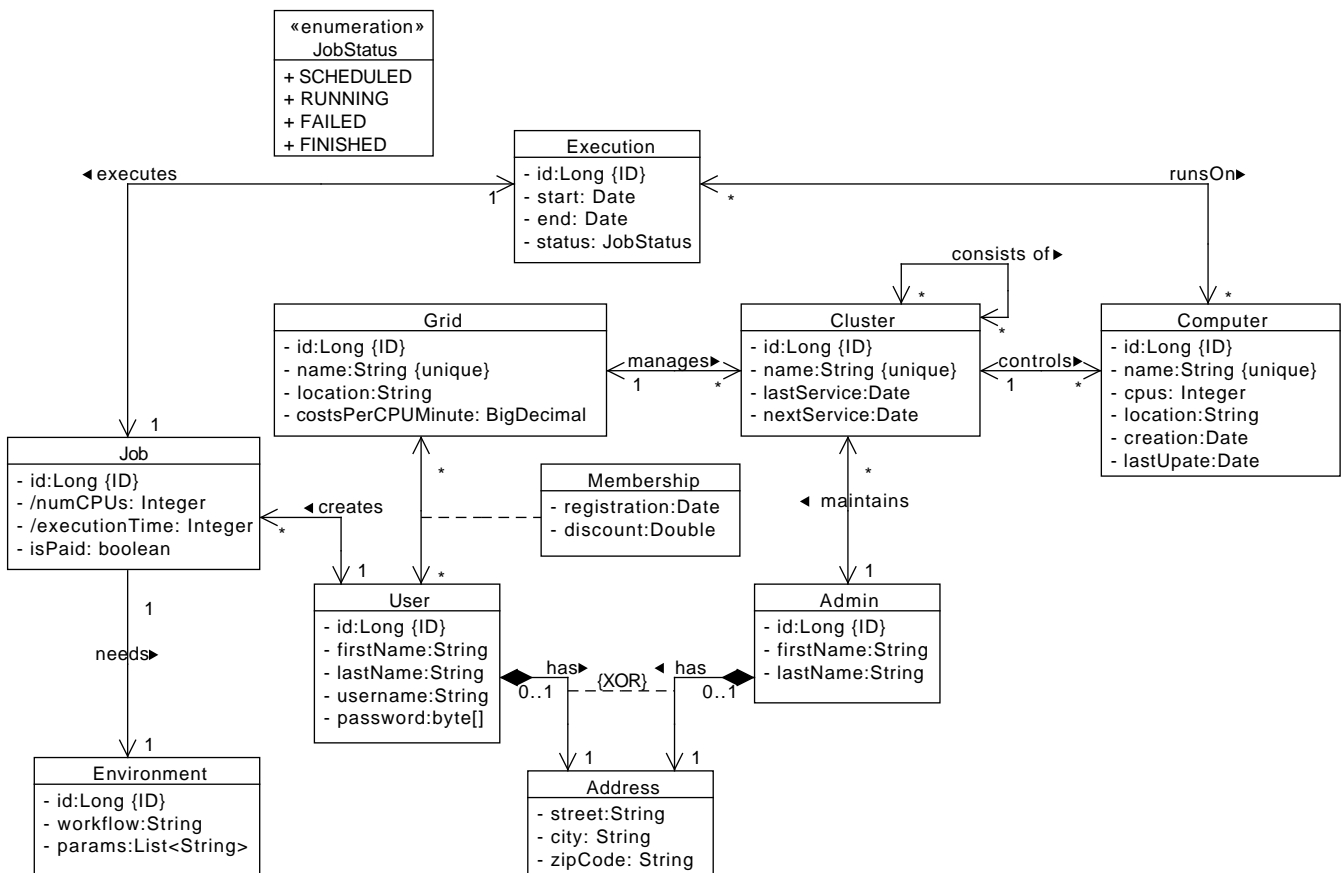


Figure 1: Grid - Domain Model

Note that each user and each admin has an address. These addresses shall be mapped as embedded classes. Also notice that each job needs some kind of environment. The environment consists of a workflow and a list of parameters. Obviously, the ordering of the parameters in this list is important,

i.e., the order should not change after persisting and loading the entity. Jobs are executed on a number of computers in parallel (a job runs on many computers, but at all at the same time). Users don't need to be members of grids to assign jobs. Memberships are identified via the grid's and the user's ID. If the user has a membership, the grid also might give discounts (in percent) to users for assigned jobs. Finally, the password of users is stored as an MD5 hash of a password string (you should never store plain text passwords to databases!). MD5 hashes are fixed size, so make sure that Hibernate uses the optimal data type for your password column.

1.a. Basic Mapping

Map these classes and all associations using the Hibernate JPA implementation (i.e., only make use of the package `javax.persistence.*` in your mappings and the respective test code). For all classes, use annotations for the mappings. The only entity you should use XML mappings for is the **Computer** entity. Make sure that you implement navigation in the data model as specified in the class diagram (for instance, the job-execution association is bidirectional, so implement it that way). Put your model code into the package `dst1.model` of your solution.

Note that you will also need your solution for this part for almost all tasks in all later assignments, so we strongly recommend that you solve at least this part of the assignment.

1.b. Inheritance

While solving task (1.a.), you may have noticed that users and admins resemble each other in many ways. Therefore, we will now implement an inheritance relationship via the abstract entity *Person*. Expand and adapt your code from task (1.a.) by choosing one of the three well-known inheritance patterns. Be prepared to argue your choice during the practice lesson! At this point, we also add information about the user bank account to be able to debit money for the processed jobs. The required changes are shown in the following diagram. You also need to implement the two new unique constraints, as well as the new not-null constraint.

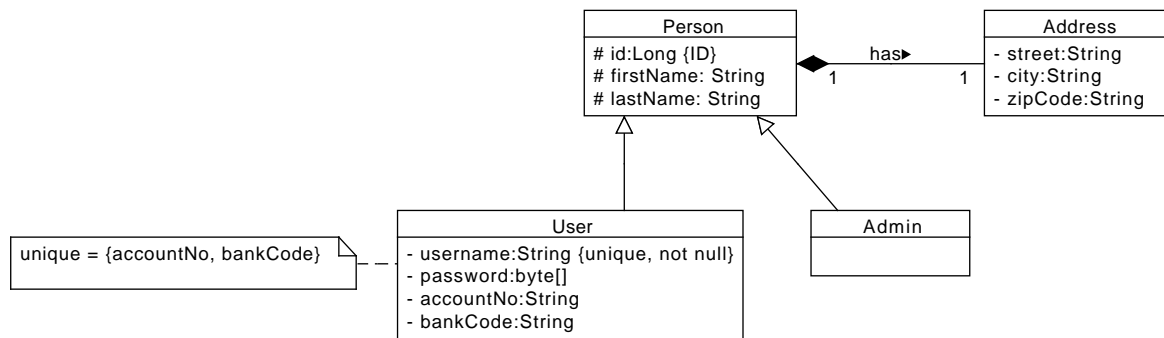


Figure 2: Grid - Inheritance Model

Now that you have finished mapping the grid's domain, you need to test/demonstrate your mapping. Store a reasonable amount of entities in the database (one or two are not enough), retrieve them, update some of them, and delete some of them again. Show that your solution works in its entirety, that is, no unnecessary tables get created, no unintended information is lost when deleting entities (e.g., because of unexpected cascades), and, of course, that there are no exceptions thrown when doing any of it. Put the resulting code into `dst1.Main#1()`.

2. JPA-QL and Hibernate Criteria API (8 Points)

- 2.a.** Write the following **Named Queries** (no plain SQL, and no inline queries!) for the domain model implemented in Task 1. Your queries should solely accept the specified parameters and return the requested types. Put your test code into `dst1.Main#2a()`.
- Find all users who have an active membership for a specific grid and assigned at least x jobs to it (given the name of the grid and the number of jobs).
 - Find the most active user(s), i.e., the user(s) who assigned the largest amount of jobs.
- 2.b.** Find all computers in Vienna (location field starts with 'AUT-VIE') and, for each of them, calculate the total usage (sum of end date minus start date for all executions of this computer). You do not need to find a single query to solve this task (you can use a combination of Java code and named queries), but you have to keep ORM-performance in mind, i.e., make sure that your solution is also reasonably fast if you have many executions on many computers. During the presentation you should be able to answer what problems you can run across here. Put the resulting code into `dst1.Main#2b()`.
- 2.c.** Write the following queries using the **Hibernate Criteria API** (i.e., you are not able to conform to JPA this time). All parameters should be optional. Put the code into the package `dst1.query` and provide some tests in `dst1.Main#2c()`.
- Find all jobs that were created by a specified user which execute a given workflow (user is given by his username, and the workflow as workflow name).
 - Use **Query-by-Example** to find all jobs with JobStatus `FINISHED` and a specified start and finish date.
- 2.d.** Now we want to optimize the following query by adding an index with Hibernate:
- `"Select u from User u where u.password = MD5(?)"`
- Assume that the user base will be very large, and this query will be executed often (i.e., whenever a user logs in). Assign the correct index value to speed up this query. Prepare to argue during the practice lessons, in which cases this index is helpful, and when it might even be harmful to the performance of the application.

3. Bean Validation (3 Points)

The Bean Validation API is a new feature in Java EE 6. You can use it to introduce restrictions on class fields or complete class instances, e.g. to verify that a certain String contains a well-formed email-address or that an Integer field has a value between two specified constants. Bean Validation becomes especially useful when validating user input before storing it to the database, but it is explicitly not tied to any programming model or application layer.

- 3.a.** In a first step we will start by applying some built-in constraints of the **Bean Validation API** to our computer (this time, you are allowed to use annotations for the computer entity):
- A computer has dates for creation and lastUpdate, both have to be in the past.
 - Every computer has a name not shorter than 5 and not longer than 25 characters.
 - The location of the computer starts with three upper-case alphabetic characters (for the Country) followed by a '-', another 3 upper-case alphabetic characters (for the City) followed by a '@' and at least 4 digits (for the zip-code). Example: 'AUT-VIE@1040'.
- 3.b.** The grid only uses Computers which have a specific amount of CPUs. Therefore, provide a constraint annotation and a constraint validator to assure that the amount of CPUs is between a defined upper and lower bound. If validation fails, return meaningful error messages. Put the code into `dst1.validator`. The annotation should look like this:

`@CPUs(min = 4, max = 8)`

In **dst1.Main#3()**, create both a valid and an invalid instance for computers. Invalid instances should violate every single constraint. Use a `javax.validation.Validator` to validate each instance and print out every existing error message.

4. Entity Manager, Entity Listeners and Interceptor (5 Points)

4.a. Entity Manager - Lifecycle

Up to this point, you already made some experiences storing and retrieving entities using the `EntityManager`. Now it is time to study the lifecycle of Persistent Entities. Create a transaction sequence to illustrate the persistence lifecycle of a job object (there has to be an association with a user). Additionally, make some source code comments that describe the current state of the activity object. Put the resulting code into **dst1.Main#4a()**. Make sure that all possible states are covered by your example.

4.b. Entity Listener

Entity Listeners allow us to register callback methods for specified lifecycle events. Provide an entity listener which sets the creation- and lastUpdate-field every time a new computer gets stored and overwrites the lastUpdate-field every time an already existing computer entity is updated. Put the code into **dst1.listener**. To show that your solution works, update some of the already persisted computer entities in **dst1.Main#4b()**.

4.c. Default Listener

Now your task is to implement a simple default listener. This listener should record the number of entities loaded, updated and removed. This listener also records how often a persist operation was successfully called and how much time all store-operations took together. Make the listener thread-safe and put its code into **dst1.listener**. Print out the results in **dst1.Main#4c()**, including the computed average time for a single persist-operation. Here is an example of how the output could look like:

Load-Operations	: 30
Update-Operations	: 20
Remove-Operations	: 15
Persist-Operations	: 40
Overall time to persist	: 187 ms
Average time to persist	: 4.561 ms

4.d. Interceptor

Task **dst1.Main#2b()** aims at minimizing the number of selects that are issued. Now we will use the Hibernate Interceptor interface to count the number of database select statements that Hibernate actually uses in the background. The Hibernate Interceptor interface provides callbacks from the session to the application, allowing the application to inspect and/or manipulate properties for a specific event. Implement the provided **dst1.interceptor.SQLInterceptor** and count the number of selects for computers and executions. It is ok to check if the SQL-String starts with the respective select-statement. Use this interceptor to validate that your solution for Task **dst1.Main#2b()** works efficiently.

In addition, you should provide methods to reset the select statement counter and print a log message. Here is an example of what this log could look like:

`Counted select statements for Computers and Executes : 10`

Test your code in **dst1.Main#4d()** by calling your previously made task **dst1.Main#2b()** and print the log message afterwards. Don't forget to reset your counter before calling task 2b(), since the interceptor will be invoked for each SQL-statement Hibernate generates.

5. NoSQL Database (8 Points)

So far, all tasks made use of a traditional SQL database (MySQL) and JPA/Hibernate. For the last practical tasks of this assignment, we will look at a different database paradigm, so-called NoSQL (Not only SQL) databases. Hence, if you have not yet done so, download and install MongoDB (version 2.0.2) from the project's web page. Use the default port, and skip authentication (evidently, you would not want to do this in production, but for our tests this is the easiest way to get started).

5.a Storing workflow results in MongoDB

So far, all data that we wanted to store in our Grid management system was fairly well-structured and uniform. However, assume now that we also need to capture and store the output produced by Grid workflows. This data is evidently quite unstructured, as different types of workflows will surely produce different types of output. For instance, a bioinformatics workflow might produce some representation of chromosome data, while a mathematical workflow might deliver some matrices as result. Furthermore, there is no way at design time to know which data we will need to store. Hence, we will use MongoDB, a schema-free NoSQL database for this part of the system.

Every workflow output can be represented as a JSON document with 2 or more properties. The first property is **job_id**, which refers back to the relational database ID of the job that produced this output. The second property is **last_updated**, containing a UNIX timestamp when this document was last changed. Finally, workflow outputs can have 0 or more additional properties, containing the actual payload of the document. An example is provided below.

```
{
  "job_id" : 2 ,
  "last_updated" : 1329469221971,
  "result_matrix" : {
    "matrix" : [ [ 1,0,0,0] , [0,1,0,0] , [0,0,1,0] , [0,0,0,1]]
  } ,
  "type" : "identity_matrix"
}
```

Your first task is to write a method **dst1.Main#5a()**, in which you add an output document for every finished job to your MongoDB installation. Add at least 5 different, not entirely trivial and at least partly structurally different, documents. Use the official MongoDB Java driver (contained in the libs folder of the template). To get you started, the provided template contains 3 example documents in JSON notation. Furthermore, as we will often retrieve documents by job id, you should add an index to speed up such queries.

5.b Querying workflow results

MongoDB primarily uses the query-by-example pattern for retrieving data, which we have already used in Task 2.c. In method **dst1.Main#5b()**, you have to write two simple queries, which (1) fetch the result document for a given job id (e.g., for the job id 1) and (2) fetch all results with a last updated field later than 1325397600. For the second query, use a filter to retrieve only the actual payload of the document (no other properties should be contained in the delivered result). Print all results of your queries to the system output stream.

5.c Map/Reduce queries

In addition to query-by-example, we can also use the more powerful, but also more complex, map/reduce pattern for data retrieval and processing. MongoDB supports map/reduce via JavaScript, that

is, you will need to write both the map and the reduce functions as JavaScript functions that operate on your stored documents. You can find a good example online⁹. Another blog post shows how you can call map/reduce via the Java driver¹⁰.

Your task is now to extend method `dst1.Main#5c()` to use map/reduce for reporting some rudimentary statistics about the stored workflow output documents. The output of your map/reduce query should be a list of all existing (top-level) payload document properties, along with the number of documents that they occur in. Do not report the properties containing the job id or the last updated field. Similarly, skip the `_id` field which MongoDB automatically generates for each document. Essentially, this report allows us to reason about how many documents there are for each type of workflow. See below for an example output. As before, print the results of your query to the system output stream.

```
{ " _id" : "alignment_block" , "value" : 3}
{ " _id" : "logs" , "value" : 1}
{ " _id" : "result_matrix" , "value" : 4}
{ " _id" : "type" , "value" : 4}
```

B. Theory Part

The following questions will be discussed during the practice lesson. At the beginning of the each lesson we hand out a list where you can specify which questions you have prepared and are willing to present. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a strong answer, you will lose **all** points for the theory part of this assignment.

5. Annotations vs. XML (1 Point)

In the previous tasks you already made some experiences using annotations and XML. What are the pros and cons of each approach, when would you use which one? Answering these questions, also keep in mind maintainability and the different roles usually involved in software projects.

6. Versioning (1 Point)

JPA provides a feature called versioning. Why and under which circumstances can this feature be useful? Think about a situation where optimistic locking may result in an (desired) exception.

7. Read-Locks (2 Point)

The EntityManager allows the programmer to set Read-locks on specified objects. What are the consequences on concurrent threads when one thread sets such a lock? Think about use-cases this behaviour may be adequate for. What problems can arise?

8. Database Isolation-Levels (2 Points)

Have a look at the different isolation-levels modern databases provide. What kind of problems might occur due to concurrency issues at what kind of level? Is it really necessary to protect your application against every type of flaw?

⁹<http://blog.fiesta.cc/post/10980328832/walkthrough-a-mongodb-map-reduce-job>

¹⁰http://blog.evilmoneylabs.com/2011/02/28/MongoDB-1_8-MR-Java/

9. Database Indices (2 Point)

What is the purpose and functioning of a database index? Using which data structures do database management systems typically store an index internally, and what are important characteristics of these data structures? What is the basic tradeoff of using an index, what are its limitations? Think of two concrete examples - one in which an index leads to an improvement and one in which the index is useless (i.e., does not lead to an improvement).

10. NoSQL Databases (2 Point)

What general types of NoSQL databases exist? Name prominent examples for each type of database, and argue when you should be using them (and also, when you should specifically *not* use them).